Functions in C++

4

Key Concepts

Return types in main( ) | Function prototyping | Call by reference | Call by value | Return by reference | Inline functions | Default arguments | Constant arguments | Function overloading

4.1 Introduction

We know that functions play an important role in C program development. Dividing a program into functions is one of the major principles of top-down, structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

Recall that we have used a syntax similar to the following in developing C programs.

```
void show( );              /* Function declaration */
main( )
{
      . . . . .
      show( );              /* Function call */
      . . . . .
}
void show( )                /* Function definition */
{
      . . . . .
      . . . . .             /* Function body */
      . . . . .
}
```

When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed and control returns to the main program when the closing brace is encountered. C++ is no exception.

Functions continue to be the building blocks of C++ programs. In fact, C++ has added many new features to functions to make them more reliable and flexible. Like C++ operators, a C++ function can be overloaded to make it perform different tasks depending on the arguments passed to it. Most of these modifications are aimed at meeting the requirements of object-oriented facilities.

In this chapter, we shall briefly discuss the various new features that are added to C++ functions and their implementation.

4.2 The Main Function

C does not specify any return type for the main( ) function which is the starting point for the execution of a program. The definition of main( ) would look like this:

```
main( )
{
      // main program statements
}
```

This is perfectly valid because the main( ) in C does not return any value.

In C++, the main( ) returns a value of type int to the operating system. C++, therefore, explicitly defines main( ) as matching one of the following prototypes:

```
int main( );
int main(int argc, char * argv[]);
```

The functions that have a return value should use the return statement for termination. The main( ) function in C++ is, therefore, defined as follows:

```
int main( )
{
      . . . . . .
      . . . . . .
      return 0;
}
```

Since the return type of functions is int by default, the keyword int in the main( ) header is optional. Most C++ compilers will generate an error or warning if there is no return statement. Turbo C++ issues the warning

```
Function should return a value
```

and then proceeds to compile the program. It is good programming practice to actually return a value from main( ).

Many operating systems test the return value (called exit value) to determine if there is any problem. The normal convention is that an exit value of zero means the program ran successfully, while a nonzero value means there was a problem. The explicit use of a return(0) statement will indicate that the program was successfully executed.

4.3 Function Prototyping

Function *prototyping* is one of the major improvements added to C++ functions . The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values. With function prototyping, a *template* is always used when declaring and defining a function. When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return value is treated correctly. Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself. These checks and controls did not exist in the conventional C functions.

Remember, C also uses prototyping. But it was introduced first in C++ by Stroustrup and the success of this feature inspired the ANSI C committee to adopt it. However, there is a major difference in prototyping between C and C++. While C++ makes the prototyping essential, ANSI C makes it optional, perhaps, to preserve the compatibility with classic C.

Function prototype is a *declaration statement* in the calling program and is of the following form:

type function-name (argument-list);

The *argument-list* contains the types and names of arguments that must be passed to the function.

Example:

float volume(int x, float y, float z);

Note that each argument variable must be declared independently inside the parentheses. That is, a combined declaration like

float volume(int x, float y, z); is illegal.

In a function declaration, the names of the arguments are *dummy* variables and therefore , they are optional. That is, the form

float volume(int, float, float);

is acceptable at the place of declaration. At this stage, the compiler only checks for the type of arguments when the function is called.

In general, we can either include or exclude the variable names in the argument list of prototypes. The variable names in the prototype just act as placeholders and, therefore, if names are used, they don't have to match the names used in the *function call or function definition.*

In the function definition, names are required because the arguments must be referenced inside the function. Example:

```
float volume(int a,float b,float c)
{
      float v - a*b*c;
      . . . . .
      . . . . .
}
```

The function volume( ) can be invoked in a program as follows:

```
float cube1 - volume(b1,w1,h1); // Function call
```

The variable *b1, w1,* and *h1* are known as the actual parameters which specify the dimensions of cube1. Their types (which have been declared earlier) should match with the types declared in the prototype. Remember, the calling statement should not include type names in the argument list.

We can also declare a function with an *empty argument list,* as in the following example:

```
void display ( );
```

In C++, this means that the function does not pass any parameters. It is identical to the statement

```
void display(void);
```

However, in C, an empty parentheses implies any number of arguments. That is, we have foregone prototyping. A C++ function can also have an 'open' parameter list by the use of ellipses in the prototype as shown below:

```
void do_something (. . .) ;
```

4.4 Call by Reference

In traditional C, a function call passes arguments by value. The *called function* creates a new set of variables and copies the values of arguments into them. The function does not have access to the actual variables in the calling program and can only work on the copies of values. This mechanism is fine if the function does not need to alter the values of the original variables in the calling program. But, there may arise situations where we would like to change the values of variables in the *calling program.* For example, in bubble sort, we compare two adjacent elements in the list and interchange their values if the first element is greater than the second. If a function is used for *bubble sort,* then it should be able to alter the values of variables in the calling function, which is not possible if the call- by-value method is used.

Provision of the *reference variables* in C++ permits us to pass parameters to the functions by reference. When we pass arguments by reference, the 'formal' arguments in the called function become aliases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data. Consider the following function:

```
void swap(int &a,int &b)          // a and b are reference variables
{
      int t - a;                  // Dynamic initialization
      a - b;
      b - t;
}
```

Now, if *m* and *n* are two integer variables, then the function call

```
swap(m, n);
```

will exchange the values of *m* and *n* using their aliases (reference variables) *a* and b. Reference variables have been discussed in detail in Chapter 3. In traditional C, this is accomplished using *pointers*and *indirection* as follows:

```
void swap1(int *a, int *b)        /* Function definition */
{
      int t;
      t - *a;                     /* assign the value at address a to t */
      *a - *b;                    /* put the value at b into a */
      *b - t;                     /* put the value at t into b */
}
```

This function can be called as follows:

```
swap1(&x, &y);              /* call by passing */
                            /* addresses of variables */
```

This approach is also acceptable in C++. Note that the call-by-reference method is neater in its approach.

4.5 Return by Reference

A function can also return a reference. Consider the following function:

```
int & max(int &x, int &y)
{
      if (x > y)
         return x;
      else
         return y;
}
```

Since the return type of max( ) is int &, the function returns reference to x or y (and not the values). Then a function call such as

max(a, b) will yield a reference to either a or b depending on their values. This means that this function call can appear on the left-hand side of an assignment statement. That is, the statement

```
max(a,b) - -1;
```

is legal and assigns -1 to a if it is larger, otherwise -1 to b.

4.6 InlineFunctions

One of the objectives of using functions in a program is to save some memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.

One solution to this problem is to use macro definitions, popularly known as *macros.* Preprocessor macros are popular in C. The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.

C++ has a different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called *inline function.* An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code (something similar to macros expansion). The inline functions are defined as follows:

```
inline function-header
{
    function body
}
```

Example:

```
inline double cube(double a)
{
      return(a*a*a);
}
```

The above inline function can be invoked by statements like

```
c - cube(3.0);
d - cube(2.5+1.5);
```

On the execution of these statements, the values of c and d will be 27 and 64 respectively. If the arguments are expressions such as 2.5 + 1.5, the function passes the value of the expression, 4 in this case. This makes the inline feature far superior to macros.

It is easy to make a function inline. All we need to do is to prefix the keyword inline to the function definition. All inline functions must be defined before they are called.

We should exercise care before making a function inline. The speed benefits of inline functions diminish as the function grows in size. At some point the overhead of the function call becomes small compared to the execution of the function, and the benefits of inline functions may be lost. In such cases, the use of normal functions will be more meaningful. Usually, the functions are made inline when they are small enough to be defined in one or two lines. Example:

```
inline double cube(double a) {return(a*a*a);}
```

Remember that the inline keyword merely sends a request, not a command, to the compiler. The compiler may ignore this request if the function definition is too long or too complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a switch, or a goto exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain static variables.
4. If inline functions are recursive.

NOTE: ***Inline*** *expansion makes a program run faster because the overhead of a function call and return is eliminated. However, it makes the program to take up more memory because the statements that define the inline function are reproduced at each point where the function is called. So, a trade-off becomes necessary.*

Program 4.1 illustrates the use of inline functions.

Program 4.1 Inline Functions

```
#include <iostream>
using namespace std;
      inline float mul(float x, float y)
      {
            return(x*y);
      }
      inline double div(double p, double q)
      {
            return(p/q);
      }
int main( )
{
      float a - 12.345;
      float b - 9.82;
```

```
    cout << mul(a,b) << "\n";
    cout << div(a,b) << "\n";
    return 0;
```

The output of Program 4.1 would be:

```
121.228
1.25713
```

4.7 Default Arguments

C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a *default value* to the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values. Here is an example of a prototype (i.e., function declaration) with default values:

```
float amount(float principal,int period,float rate-0.15);
```

The default value is specified in a manner syntactically similar to a variable initialization. The above prototype declares a default value of 0.15 to the argument rate. A subsequent function call like

```
value - amount(5000,7);          // one argument missing
```

passes the value of 5000 to principal and 7 to period and then lets the function use default value of 0. 15 for rate. The call

```
value - amount(5000,5,0.12);          // no missing argument
```

passes an explicit value of 0.12 to rate.

A default argument is checked for type at the time of declaration and evaluated at the time of call. One important point to note is that only the trailing arguments can have default values and therefore we must add defaults from *right to left.* We cannot provide a default value to a particular argument in the middle of an argument list. Some examples of function declaration with default values are:

```
int mul(int i, int j-5, int k-10);          // legal
int mul(int i-5, int j);                    // illegal
int mul(int i-0, int j, int k-10);          // illegal
int mul(int i-2, int j-5, int k-10);        // legal
```

Default arguments are useful in situations where some arguments always have the same value. For instance, bank interest may remain the same for all customers for a particular period of deposit. It also provides a greater flexibility to the programmers. A function can be written with more parameters than are required for its most common application. Using default arguments, a programmer can use only those arguments that are meaningful to a particular situation. Program 4.2 illustrates the use of default arguments.

Program 4.2 Default Arguments

```
#include<iostream>
#include<conio.h>

using namespace std;

int main( )
{
      float amount;
float value(float p, int n, float r-0.15); //prototype
void printline(char ch-'*',int len-40); //prototype
printline( );                  //use default values for arguments
```

```
    amount - value (5000.00,5);              //default for 3rd
argument
    cout<<"\n Final Value - "<<amount<<"\n\n";

    amount - value (10000.00,5,0.30); //pass all arguments
explicitly

    cout<<"\n Final Value - "<<amount<<"\n\n";

    printline('-');                  //use default value for second
argument

    getch ( );

    return 0;
    }
    float value (float p, int n, float r)
    {
        int year - 1; float sum - p;

        while(year<-n)
        {
            sum - sum*(1+r);
            year - year+1;
        }
        return (sum);
        }
    void printline(char ch, int len)
    {
        for (int i-1;i<-len;i++)
        printf("%c",ch);
        printf("\n");
    }
```

The output of Program 4.2 would be:

```
**********************************
Final Value - 10056.8
Final Value - 37129.3
-----------------------------------------------
```

Advantages of providing the default arguments are:

1. We can use default arguments to add new parameters to the existing functions.
2. Default arguments can be used to combine similar functions into one.

4.8 const Arguments

In C++, an argument to a function can be declared as const as shown below.

```
int strlen(const char *p);
int length(const string &s);
```

The qualifier **const** tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

4.9 Recursion

Recursion is a situation where a function calls itself meaning, one of the statements in the function definition makes a call to the same function in which it is present. It may sound like an infinite looping condition but just as a loop has a conditional check to take the program control out of the loop, a recursive function also possesses a base case which returns the program control from the current instance of the function to call back to the calling function. For

example, in a series of recursive calls to compute the factorial of a number, the base case would be a situation where factorial of 0 is to be computed. Let us consider a few examples (Program 4.3 and 4.4) to understand how the recursive approach works.

Program 4.3 Calculating Factorial of a Number

```
#include <iostream>
#include <conio.h> using namespace std;

long fact(int n)
{
      if (n -- 0)       //base case
      return 1;

      return (n * fact(n-1));       //recursive function call
}
int main( )
{
      int num;

      cout << "Enter a positive integer: ";
      cin >> num;

      cout << "Factorial of " << num << " is " << fact(num);

      getch ( );
      return 0;
}
```

The output of program 4.3 would be:

```
Enter a positive integer: 10
Factorial of 10 is 3628800
```

Program 4.5 Solving Tower of Hanoi Problem

```
#include <iostream>
#include <conio.h>
using namespace std;

void TOH(int d,char tower1, char tower2, char tower3)
{
      if (d--1) //base case
      {
            cout << "\nShift top disk from tower " << tower1 << "
to tower " << tower2;
            return;
      }
      TOH(d-1,tower1, tower3, tower2); //recursive function call
cout << "\nShift top disk from tower " << tower1 << " to tower " <<
tower2;

      TOH(d-1,tower3, tower2, tower1); //recursive function call
}
int main( )
{
      int disk;

      cout << "Enter the number of disks: ";
      cin >> disk;

      if (disk < 1)
            cout << "\nThere are no disks to shift";
      else
            cout <<"\nThere are " << disk << " disks in tower 1\n";
```

```
            TOH(disk, '1', '2', '3 ' );
            cout << "\n\n" << disk << " disks in tower 1 are shifted to
tower 2";

            getch( );
            return 0;
      }
```

The output of Program 4.4 would be:

```
Enter the number of disks: 3
There are 3 disks in tower 1
Shift top disk from tower 1 to tower 2
Shift top disk from tower 1 to tower 3
Shift top disk from tower 2 to tower 3
Shift top disk from tower 1 to tower 2
Shift top disk from tower 3 to tower 1
Shift top disk from tower 3 to tower 2
Shift top disk from tower 1 to tower 2
3 disks in tower 1 are shifted to tower 2
```

4.10 FunctionOverloading

As stated earlier, *overloading* refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as *function polymorphism* in OOP.

Using the concept of function overloading; we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but

not on the function type. For example, an overloaded add( ) function handles different types of data as shown below:

```
// Declarations
int add(int a, int b);                          // prototype 1
int add(int a, int b, int c);                   // prototype 2
double add(double x, double y);                 // prototype 3
double add(int p, double q);                    // prototype 4
double add(double p, int q);                    // prototype 5

// Function calls
cout << addi ;5, 10);                           // uses prototype 1
cout << add ( 15, 10.0);                        // uses prototype 4
cout << add( 12.5, 7.5);                        // uses prototype 3
cout << add( 5, 10, 15);                        // uses prototype 2
cout << add( 0.75, 5);                          // uses prototype 5
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

   char to int
   float to double

   to find a match.

3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique.

If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the

following two functions:

```
long square(long n)
double square(double x)
```

A function call such as

```
square(10)
```

will cause an error because **int** argument can be converted to either **long** or **double**, thereby creating an ambiguous situation as to which version of **square( )** should be used.

4. If all of the steps fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.

Program 4.5 illustrates function overloading.

Program 4.5 Function Overloading

```
// Function volume( ) is overloaded three times
#include <iostream>

using namespace std;

// Declarations (prototypes)

int volume(int);
double volume(double, int);
long volume(long, int, int);

int main( )
{
```

```
        cout<<“Calling the volume( ) function for computing the
volume of a cube - “<<volume(10)<<“\n”;
        cout<<“Calling the volume( ) function for computing the
volume of a cylinder - “<<volume(2.5,8)<<“\n”;
        cout<<“Calling the volume( ) function for computing the
volume of a rectangular box - “<<volume(100L,75,15); return 0;
    }

    // Function definitions

    int volume(int s) // cube
    {
        return(s* s* s);
    }
    double volume(double r, int h) // cylinder
    {
        return(3.14519*r*r*h);
    }
    long volume(long l, int b, int h) // rectangular box
    {
        return (l*b*h);
    }
```

The output of Program 4.5 would be:

```
    Calling the volume( ) function for computing the volume of a
cube - 1000
    Calling the volume( ) function for computing the volume of a
cylinder - 157.26
    Calling the volume( ) function for computing the volume of a
rectangular box - 112500
```

Overloading of the functions should be done with caution. We should not overload unrelated functions and should reserve function overloading for functions that perform closely related tasks.

Sometimes, the default arguments may be used instead of overloading. This may reduce the number of functions to be defined.

Overloaded functions are extensively used for handling class objects. They will be illustrated later when the classes are discussed in the next chapter.

4.11 Friend and Virtual Functions

C++ introduces two new types of functions, namely, friend function and virtual function. They are basically introduced to handle some specific tasks related to class objects. Therefore, discussions on these functions have been reserved until after the class objects are discussed. The friend functions are discussed in Sec. 5.15 of the next chapter and virtual functions in Sec. 9.5 of Chapter 9.

4.12 Math Library Functions

The standard C++ supports many math functions that can be used for performing certain commonly used calculations. Most frequently used math library functions are summarized in Table 4.1.

Table 4.1 *Commonly used math library functions*

NOTE: *The argument variables **x** and **y** are of type **double** and all the functions return the data type **double**.*

To use the math library functions, we must include the header file **math.h** in conventional C++ and **cmath** in ANSI C++.

Program 4.6

Program 4.6 demonstrates the use of math functions:

Program 4.6 Use of Math Functions

```
#include<iostream>
#include<conio.h>
#include<iomanip>
#include<math.h>

using namespace std;

int main( )
{

        cout<<fixed<<setprecision(2);
        cout<<“sin (100) - “<<sin(100.00)<<“\n”; //Computing sine
value
        cout<<“cos (100) - “<<cos(100.00)<<“\n”; //Computing
cosine value
        cout<<“tan (100) - “<<tan(100.00)<<“\n”; //Computing
tangent value
        cout<<“7 to the power of 6 - “<<pow(7.0,6.0)<<“\n”;
//Computing power of one value raised to the other
        cout<<“log10 (10) - “<<log10(10.00)<<“\n”; //Computing
log to the base of 10 value
        cout<<“sqrt (2) - “<<sqrt(2.00)<<“\n”; //Computing the
square root of a value

        getch( );
        return 0;
}
```

The output of Program 4.6 would be:

sin (100) - -0.51
cos (100) - 0.86
tan (100) - -0.59
7 to the power of 6 - 117649.00
log10 (10) - 1.00
sqrt (2) - 1.41

Summary

- It is possible to reduce the size of program by calling and using functions at different places in the program.
- In C++ the main( ) returns a value of type **int** to the operating system. Since the return type of functions is **int** by default, the keyword **int** in the main( ) header is optional. Most C++ compilers issue a warning, if there is no return statement.
- Function prototyping gives the compiler the details about the functions such as the number and types of arguments and the type of return values.
- Reference variables in C++ permit us to pass parameters to the functions by reference. A function can also return a reference to a variable.
- When a function is declared inline the compiler replaces the function call with the respective function code. Normally, a small size function is made as **inline.**
- The compiler may ignore the inline declaration if the function declaration is too long or too complicated and hence compile the function as a normal function.
- C++ allows us to assign default values to the function parameters when the function is declared. In such a case we can call a function without specifying all its arguments. The defaults are always added from right to left.

- In C++, an argument to a function can be declared as **const,**indicating that the function should not modify the argument.
- C++ allows function overloading. That is, we can have more than one function with the same name in our program. The compiler matches the function call with the exact function code by checking the number and type of the arguments.
- C++ supports two new types of functions, namely **friend** functions and **virtual** functions.
- Many mathematical computations can be carried out using the library functions supported by the C++ standard library.

Key Terms

actual arguments | argument l ist | bubble sort | call by reference | call by value | called function | calling program | calling statement | cmath | const arguments | declaration statement | default arguments | default values | dummy variables | ellipses | empty argument list | exit value | formal arguments | friend functions | function call | function definition | function overloading | function polymorphism | function prototype | indirection | inline | inline functions | macros | **main( )** | math l ibrary | **math.h** | overloading | pointers | polymorphism | prototyping | reference variable | return by reference | **return** statement | return type | **return( )** | template | virtual functions

Review Questions

4.1 State whether the following statements are TRUE or FALSE.

(a) A function argument is a value returned by the function to the calling program.

(b) When arguments are passed by value, the function works with the original arguments in the calling program.

(c) When a function returns a value, the entire function call can be assigned to a variable.

(d) A function can return a value by reference.

(e) When an argument is passed by reference, a temporary variable is created in the calling program to hold the argument value.

(f) It is not necessary to specify the variable name in the function prototype.

4.2 Describe the different styles of writing prototypes.

4.3 Find errors, if any, in the following function prototypes.

(a) float average(x,y);

(b) int mul(int a,b);

(c) int display(...);

(d) void Vect(int? &V, int & size);

(e) void print(float data[ ], size - 20);

4.4 What is the main advantage of passing arguments by reference?

4.5 What is the most significant advantage that you see in using references instead of pointers?

4.6 When will you make a function inline? Why?

4.7 How does an inline function differ from a preprocessor macro?

4.8 When do we need to use default arguments in a function?

4.9 What is the significance of an empty parenthesis in a function declaration?

4.10 What do we mean by overloading of a function? When do we use this concept?

4.11 How do we achieve function overloading? On what basis, the compiler distinguishes between a set of overloaded functions having the same name?

4.12 Comment on the following function definitions:

```
(a) int *f( )
{
        int m - 1;
        . . . .
        . . . .
        return(&m);
}

(b) double f( )
{
        . . . .
        . . . .
        return(1);
}

(c) int & f( )
{
        int n - 10;
        . . . .
        . . . .
        return(n);
}
```

Debugging Exercises

4.1 Identify the error in the following program.

```
#include <iostream.h>
int fun( )
{
        return 1;
}
        float fun( )
{
        return 10.23;
void main( )
{
        cout << (int)fun( ) << ‘ ‘;
        cout << (float)fun( ) << ‘ ‘;
}
```

4.2 Identify the error in the following program.

```
#include <iostream.h>

void display(const int const1-5)
{
        const int const2-5;
        int array1[const1];
        int array2[const2];
        for(int i-0; i<5; i++)
        {
                array1[i] - i;
                array2[i] - i*10;
                cout << array1[i] << ‘ ‘ << array2[i] <<‘ ‘,
        }
}
void main( )
{
```

```
        display(5);
}
```

4.3 Identify the error in the following program.

```
#include <iostream.h>
int gValue-10;
void extra( )
{
        cout << gValue << ‘ ‘;
}
        void main( )
{
        extra ( ) ;
        {
                int gValue - 20;
                cout << gValue << ‘ ‘;
                cout << : : gValue << ‘ ‘ ;
        }
}
```

4.4 Find errors, if any, in the following function definition for displaying a matrix: void display (int A[ ] [ ], int m, int n)

```
{
        for(i-0; i<m; i++)
        for(j-0; j<n; j++)
                cout << “ “ << A [i] [j] ;
        cout << “\n”;
}
```

Programming Exercises

4.1 Write a function to read a matrix of size m x n from the keyboard.

4.2 Write a program to read a matrix of size m x n from the keyboard and display the same on the screen using functions.

4.3 Rewrite the program of Exercise 4.2 to make the row parameter of the matrix as a default argument.

4.4 The effect of a default argument can be alternatively achieved by overloading. Discuss with an example.

4.5 Write a macro that obtains the largest of three numbers.

4.6 Redo Exercise 4.5 using inline function. Test the function using a main program.

4.7 Write a function power( ) to raise a number m to a power n. The function takes a double value for m and int value for n, and returns the result correctly. Use a default value of 2 for n to make the function to calculate squares when this argument is omitted. Write a main that gets the values of m and n from the user to test the function.

4.8 Write a function that performs the same operation as that of Exercise 4.7 but takes an int value for m. Both the functions should have the same name. Write a main that calls both the functions. Use the concept of function overloading.

4.9 Write a program to compute the area of a triangle and a circle by overloading the area( ) function.